Great Idea #1: Abstraction (Levels of Representation/Interpretation)

High Level Language Program (e.g., C)

Assembly Language Program (e.g., RISC-V)

Machine Language Program (RISC-V)

Compiler

Assembler

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

Anything can be represented as a number, i.e., data or instructions

We are here!

lw t0, t2, 0
lw t1, t2, 4
sw t1, t2, 0
sw t0, t2, 4

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
Recap: Complete RV32I ISA

| imm[31:12] | rd | 011011 | LUI |
| imm[31:12] | rd | 011011 | AUIPC |
| imm[11:0] | rd | 011011 | IALR |
| imm[11:0] | rd | 011011 | BEQ |
| imm[11:0] | rd | 011011 | BNE |
| imm[11:0] | rd | 011011 | BL |
| imm[11:0] | rd | 011011 | BCS |
| imm[11:0] | rd | 011011 | BLT |
| imm[11:0] | rd | 011011 | BLTU |
| imm[11:0] | rd | 011011 | BCSL |
| imm[11:0] | rd | 011011 | LB |
| imm[11:0] | rd | 011011 | LH |
| imm[11:0] | rd | 011011 | LW |
| imm[11:0] | rd | 011011 | LBU |
| imm[11:0] | rd | 011011 | LHU |
| imm[11:0] | rd | 011011 | SB |
| imm[11:0] | rd | 011011 | SH |
| imm[11:0] | rd | 011011 | SW |
| imm[11:0] | rd | 011011 | ADDI |
| imm[11:0] | rd | 011011 | SLTI |
| imm[11:0] | rd | 011011 | SLTIU |
| imm[11:0] | rd | 011011 | XORI |
| imm[11:0] | rd | 011011 | ORI |
| imm[11:0] | rd | 011011 | ANDI |

Not in CS61C
“State” Required by RV32I ISA

Each instruction reads and updates this state during execution:

1. **Registers** ($x0..x31$)
   - Register file (or regfile) \(\text{Reg} \) holds 32 registers x 32 bits/register: \(\text{Reg}[0]..\text{Reg}[31]\)
   - First register read specified by \(rs1\) field in instruction
   - Second register read specified by \(rs2\) field in instruction
   - Write register (destination) specified by \(rd\) field in instruction
   - \(x0\) is always 0 (writes to \(\text{Reg}[0]\) are ignored)

2. **Program Counter (PC)**
   - Holds address of current instruction

3. **Memory (MEM)**
   - Holds both instructions & data, in one 32-bit byte-addressed memory space
   - We’ll use separate memories for instructions (IMEM) and data (DMEM)
     - Later we’ll replace these with instruction and data caches
   - Instructions are read (fetched) from instruction memory (assume IMEM read-only)
   - Load/store instructions access data memory
One-Instruction-Per-Cycle RISC-V Machine

On every tick of the clock, the computer executes one instruction

1. Current state outputs drive the inputs to the combinational logic, whose outputs settles at the values of the state before the next clock edge

2. At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle
Basic Phases of Instruction Execution

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Register Write

Clock time

IMEM
alu
DMEM
Reg[]
Implementing the **add** instruction

```
add rd, rs1, rs2
```

- Instruction makes two changes to machine’s state:
  
  \[
  \text{Reg}[rd] = \text{Reg}[rs1] + \text{Reg}[rs2]
  \]
  
  \[
  \text{PC} = \text{PC} + 4
  \]
Datapath for `add`

```
| CS 61c |
```
Timing Diagram for **add**

```
add x1, x2, x3
add x6, x7, x9
```
Implementing the **sub** instruction

\[
\text{sub \ rd, rs1, rs2}
\]

- Almost the same as `add`, except now have to subtract operands instead of adding them
- `inst[30]` selects between add and subtract
**Datapath for add/sub**

**Control Logic**

- **IMEM**
  - DataA
  - AddrA
  - inst[24:20]
  - inst[19:15]
- **Reg[]**
  - AddrD
  - DataD
  - inst[11:7]
- **RegWEn**
  - (1=write, 0=no write)
- **ALU**
  - ALUSel
  - (Add=0/Sub=1)

**Connections**

- **pc+4**
- **+4**
- **IMEM**
- **Reg[]**
- **RegWEn**
- **ALU**

**Paths**

- pc → IMEM → Reg[] → ALU → ALUSel
- pc+4 → IMEM
### Implementing other R-Format instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
</tr>
</tbody>
</table>

- All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function.
Implementing the **addi** instruction

- **RISC-V Assembly Instruction:**
  
  ```
  addi  x15,x1,-50
  ```

  - **Binary Representation:**
    - **imm**: 0000110111001110
    - **rs1**: 0000000000000001
    - **rd**: 0000000000000101
    - **OP-Imm**: 00010000

  - **Decoded Values:**
    - **imm**: -50
    - **rs1**: 1
    - **rd**: 15
    - **OP-Imm**: ADD
Datapath for add/sub

Control Logic

pc+4

IMEM

inst[31:0]

RegWEn
(1=write, 0=no write)

ALUSel
(Add=0/Sub=1)

Reg[rs1]

Reg[rs2]

alu

IMEM

inst[11:7]

inst[19:15]

inst[24:20]

Reg[]

AddrA

DataA

AddrB

DataB

AddrD

DataD

pc
Adding **addi** to datapath

```
+4
pc+4

IMEM


Reg[]
DataD  AddrD
AddrA  DataA
AddrB  DataB

Imm. Gen

inst[31:20]
imm[31:0]

Reg[rs1]
Reg[rs2]

alu

ALUSel=Add

Control Logic

Controller

inst[31:0] ImmSel=I RegWEn=1 BSel=1
```
I-Format immediates

- High 12 bits of instruction (\text{inst}[31:20]) copied to low 12 bits of immediate (\text{imm}[11:0])
- Immediate is sign-extended by copying value of \text{inst}[31] to fill the upper 20 bits of the immediate value (\text{imm}[31:12])
Adding addi to datapath

Also works for all other I-format arithmetic instruction (slti, sltiu, andi, ori, xori, slli, srli, srai) just by changing ALUSel
Break!
Implementing Load Word instruction

- RISC-V Assembly Instruction: `lw x14, 8(x2)`

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imm[11:0]</code></td>
<td>010000</td>
</tr>
<tr>
<td><code>rs1</code></td>
<td>00010</td>
</tr>
<tr>
<td><code>funct3</code></td>
<td>01110</td>
</tr>
<tr>
<td><code>rd</code></td>
<td>000011</td>
</tr>
<tr>
<td><code>opcode</code></td>
<td>LOAD</td>
</tr>
</tbody>
</table>

`imm=+8  rs1=2  LW  rd=14  LOAD`
Adding addi to datapath

IMEM

pc+4

pc

+4

Reg[]

DataD

AddrD

inst[11:7]

inst[19:15]

inst[24:20]

inst[24:20]

imm[31:0]

Reg[rs1]

Reg[rs2]

IMEM

inst[31:20]

Imm. Gen

alu

ALU

BSel=1

RegWEn=1

ImmSel=I

ALUSel=Add

Control Logic
Adding lw to datapath
Adding `lw` to datapath

```
IMEM
 AddrD
 DataD
 AddrA
 DataA
 AddrB
 DataB

Reg[]
 AddrD
 DataD

ALU
 Addr
 DataR

DMEM
 Addr
 DataR

IMEM
 inst[11:7]
 inst[19:15]
 inst[24:20]
 inst[31:20]

Imm. Gen
 ImmSel = I
 RegWEn = 1

IMEM
 pc+4

PC

alu

DMEM
 mem

alu

wb

wb

Reg[rs1]
Reg[rs2]

0
1

0
1

B Sel = 1
ALUSel = add
MemRW = Read
WBSel = 0

inst[31:0]

inst[31:0]

Reg[rs1]
Reg[rs2]

imm[31:0]

inst[11:7]

inst[19:15]

inst[24:20]

inst[31:20]

alu

mem

wb

Reg[]

Inst[31:0]

ImmSel = I
RegWEn = 1

B Sel = 1
ALUSel = add
MemRW = Read
WBSel = 0

pc

pc+4

pc+4
```
- Supporting the narrower loads requires additional circuits to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.

funct3 field encodes size and signedness of load data
Implementing Store Word instruction

- RISC-V Assembly Instruction:
  \[ \text{sw} \ x14, \ 8 \ (x2) \]

Table representation:

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>01110</td>
<td>00010</td>
<td>010</td>
<td>01000</td>
<td>0100011</td>
</tr>
</tbody>
</table>
```

- offset[11:5] = 0
- rs2 = 14
- rs1 = 2
- SW
- offset[4:0] = 8
- combined 12-bit offset = 8
Adding \( lw \) to datapath
Adding \textbf{sw} to datapath

\begin{itemize}
  \item \textbf{IMEM}:
    \begin{itemize}
      \item \text{inst[11:7]}
      \item \text{inst[19:15]}
      \item \text{inst[24:20]}
    \end{itemize}
  \item \textbf{ALU}:
    \begin{itemize}
      \item \text{Reg[rs1]}
      \item \text{Reg[rs2]}
    \end{itemize}
  \item \textbf{DMEM}:
    \begin{itemize}
      \item \text{AddrD}
      \item \text{DataD}
    \end{itemize}
  \item \textbf{imm[31:0]}:
    \begin{itemize}
      \item \text{imm[31:0]}
    \end{itemize}
\end{itemize}

\begin{itemize}
  \item \text{Inst[31:0]}
  \item \text{ImmSel=S}
  \item \text{RegWEn=0}
  \item \text{Bsel=1}
  \item \text{ALUSel=Add}
  \item \text{MemRW=Write}
  \item \text{WBSel=*}
\end{itemize}

*= "Don't Care"
Adding `sw` to datapath

Adding `sw` to the datapath involves modifying the data flow to handle the `sw` instruction, which typically stores the contents of one memory location into another. The diagram illustrates the new connections required to accommodate this operation.

Key elements of the datapath diagram include:
- The memory module (IMEM) is connected to the program counter (pc) to fetch instructions and data.
- The ALU (Arithmetic Logic Unit) is receiving additional inputs to handle the `sw` instruction.
- The data memory (DMEM) is fed with the result of the ALU operation.
- The immediate generator (Imm. Gen) provides the immediate value needed for the `sw` instruction.

The routing of data and control signals is shown with arrows, indicating the flow of information through the components.

Annotations and labels include:
- `inst[31:0]`: Instruction word
- `ImmSel=S`: Immediate selection
- `RegWEn`: Register write enable
- `BSel=1`: Bus selection
- `ALUSel=Add`: ALU selection
- `MemRW=Write`: Memory read/write
- `WBSel=`*: Don’t Care

The diagram visually represents the integration of the `sw` instruction into the existing datapath architecture.
I-Format immediates

High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])

Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])
Just need a 5-bit mux to select between two positions where low five bits of immediate can reside in instruction
• Other bits in immediate are wired to fixed positions in instruction
Implementing Branches

• B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
• But now immediate represents values -4096 to +4094 in 2-byte increments
• The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)
Adding \textbf{sw} to datapath

- 

- IMEM

- +4

- pc

- pc+4

- inst[31:0]

- InstSel

- RegWEn

- Bsel

- ALUSel

- MemRW

- WBSel=

- DMEM

- ALU

- DataR

- DataW

- wb

- 0

- 1

- Addr

- AddrA

- AddrB

- DataA

- DataB

- DataD

- AddrD

- Reg[]
Adding branches to datapath
Adding branches to datapath
Branch Comparator

- \( \text{BrEq} = 1, \) if \( A = B \)
- \( \text{BrLT} = 1, \) if \( A < B \)
- \( \text{BrUn} = 1 \) selects unsigned comparison for \( \text{BrLT} \), \( 0 = \) signed
- \( \text{BGE} \) branch: \( A \geq B, \) if \( \neg (A < B) \)
Administrivia (1/2)

- Midterm 1 has been regraded
- Grades Available on glookup

MINIMUM       MEDIAN       MAXIMUM       MEAN       STD DEV
11.25         63.25        90.0          61.99      14.45
Administrivia (2/2)

• Project 2.2 due Friday at 11:59pm
• Project 3.1 to be released next Wednesday (2/28)
  • RISC-V implementation in “logisim”
  • Due Tuesday
• Homework 2 is released and due next Friday at 11:59pm
• No Guerrilla Session this week—will start up again next Thursday 3/1
Break!
Implementing **JALR** Instruction (I-Format)

- **JALR** \(rd, rs, immediate\)
  - Writes PC+4 to Reg[rd] (return address)
  - Sets PC = Reg[rs1] + immediate
  - Uses same immediates as arithmetic and loads
    - *no* multiplication by 2 bytes
Adding branches to datapath
Adding `jalr` to datapath
Adding `jalr` to datapath

![Datapath Diagram](image)

- **IMEM**
  - `Inst[11:7]`
  - `Inst[19:15]`
  - `Inst[24:20]`
  - `Inst[31:7]`

- **ALU**
  - `Reg[rs1]`
  - `Reg[rs2]`

- **DMEM**
  - `Addr`
  - `DataW`

- **Reg[]**
  - `DataD`
  - `AddrD`
  - `AddrA`
  - `AddrB`
  - `DataA`
  - `DataB`

- **Branch Comp.**
  - `BrUn=*`
  - `BrEq=*`
  - `BrLT=*`

- **PCSel**
  - `Inst[31:0]`
  - `ImmSel=B`
  - `RegWEn=1`

- **WBSel=2**
  - `MemRW=Read`
  - `ALUSel=Add`

- **Inst[31:0]**
  - `Bsel=1`
  - `Asel=0`
Implementing \texttt{jal} Instruction

- JAL saves PC+4 in Reg[rd] (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm2^{19}$ locations, 2 bytes apart
  - $\pm2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost
Adding jal to datapath
Adding jal to datapath
Single-Cycle RISC-V RV32I Datapath
And in Conclusion, ...

• Universal datapath
  – Capable of executing all RISC-V instructions in one cycle each
  – Not all units (hardware) used by all instructions

• 5 Phases of execution
  – IF, ID, EX, MEM, WB
  – Not all instructions are active in all phases

• Controller specifies how to execute instructions
  – what new instructions can be added with just most control?